Towards Heterogeneous and Distributed Computing in C++

Gordon Brown - Senior Software Engineer, SYCL & C++

DHPCC++ – May 2019
About me...

- Background in C++ programming models for heterogeneous systems
- Developer with Codeplay Software for 7 years
- Worked on ComputeCpp (SYCL) for 6 years
- Contributor to the Khronos SYCL standard since its inception
- Contributor to ISO C++ executors and heterogeneity for over 3 years
Contributors

Agenda

What are C++ executors?

Properties

Oneway executors

Twoway executors

Supporting affinity
Unified executor interface

Standard / proprietary libraries
- SYCL
- invoke
- define_task_block
- parallel algorithms
- future::then
- post
- Kokkos
- defer
- async
- dispatch
- asynchronous operations
- strand<>

Executors

Third-party / OS hardware abstractions
- OpenCL / CUDA / HCC / HMM
- OpenMP / MPI
- OS threads
- Boost.Asio / Networking TS

Hardware resources
auto fut = std::async(factorial, input);
auto res = fut.get();
auto fut = std::async(factorial, input);
auto res = fut.get();

auto fut = std::async(gpu_executor{}, factorial, input);
auto res = fut.get();
std::sort(par, data.begin(), data.end());
std::sort(par, data.begin(), data.end());

std::sort(par.on(gpu_executor{}), data.begin(), data.end());
• An **executor** is a light-weight object
- An **executor** is a light-weight object.
- It creates **execution agents** that invoke a callable.
• An **executor** is a light-weight object
• It creates **execution agents** that invoke a callable
• It has a number of **execution functions** which provide different way of creating **execution agents**
• An **executor** is a light-weight object
• It creates **execution agents** that invoke a callable
• It has a number of **execution functions** which provide different ways of creating **execution agents**
• It has a number of **properties** associated with it that dictate its **execution functions** and the operational semantics of the **execution agents** it creates
• An **executor** is a light-weight object
• It creates **execution agents** that invoke a callable
• It has a number of **execution functions** which provide different ways of creating **execution agents**
• It has a number of **properties** associated with it that dictate its **execution functions** and the operational semantics of the **execution agents** it creates
• It is generally associated with an **execution context**, which manages the **execution agents** it creates
Agenda

What are C++ executors?

Properties

Oneway executors

Twoway executors

Supporting affinity
● Properties provide a software abstraction for executors to express the relationship between algorithm requirements and hardware capabilities
  ○ They allow you to require that an executor support a property
  ○ They allow you to query the value of an executor property

● This facilitates better performance portability in algorithm design
  ○ Different layers of an algorithm can be specialized or adapted based on executor properties
Performing a `require` returns an executor that will have the requested properties:

- If the properties are already supported the original executor is returned.
- If the properties are not supported this will result in a compile-time error.
Performing a `prefer` returns an executor that may have the requested properties:
- If the properties are already supported the same executor is returned.
- If the properties are not supported the executor will simply return the original executor.
● Performing a query returns the current value of a specific property
  ○ In many cases this value will be a boolean
  ○ In some cases this query can be performed at compile-time if property::static_query_v is available
● Properties that are successfully requested via require or prefer can be supported in two ways
  ○ An executor implementation can natively support the property
  ○ An executor can support a property via an adaptation
oneway_executor exec;

auto newExec = require(exec, blocking.never); // Must return a non-blocking executor

auto fut = newExec.execute(func);
oneway_executor exec;
auto newExec = require(exec, blocking.never); // Must return a non-blocking executor
auto fut = newExec.execute(func);

oneway_executor exec;
auto newExec = prefer(exec, blocking.never); // May or may not return a non-blocking executor
newExec.execute(func);
```cpp
oneway_executor exec;
auto newExec = require(exec, blocking.never);  // Must return a non-blocking executor
auto fut = newExec.execute(func);

oneway_executor exec;
auto newExec = prefer(exec, blocking.never);  // May or may not return a non-blocking executor
newExec.execute(func);

oneway_executor exec;
auto newExec = prefer(exec, blocking.never);  // May or may not return a non-blocking executor
auto isNonBlocking = query(newExec, blocking.never);
```
Agenda

What are C++ executors?

Properties

Oneway executors

Twoway executors

Supporting affinity
● Oneway executors provide execution functions which execute a callable without a communication channel
  ○ Eager “Fire and forget” execution
  ○ No return value
  ○ Synchronisation and error handling are managed via another channel

● Single and bulk cardinality
  ○ Execute a callable exactly once on a single execution agent
  ○ Execute a callable in multiple iterations on multiple execution agents
oneway_executor exec;

exec.execute([&]() {
    ...
});
oneway_executor exec;
exec.execute([&]() {
    ...
});

bulk_executor exec;
exec.bulk_execute([&](index<N> i, auto r, auto s){
    ...
}, shape, resultFactory, sharedFactory);
Agenda

What are C++ executors?

Properties

Oneway executors

Twoway executors

Supporting affinity
● Twoway executors provide an execution functions which execute a callable with a communication channel
  ○ Propagates a return value or an error
  ○ Provides a predicate to later callables

● Sender/receiver model
  ○ Lazy generalization of futures and promises
    ■ Sender: lazy future
    ■ Receiver: lazy promise
  ○ Composition of nested callables
  ○ Communication channel doesn’t require shared state allocation or synchronization
Execute $f$ on the CPU, then execute $g$ on the GPU
Execute \( f \) on the CPU, then execute \( g \) on the GPU

```cpp
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
```
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});

res = invoke(f, 42)
res = invoke(g, res)
result = res
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});

res = invoke(f, 42)

value(res)

exec.schedule()

result = res

value(res)

value(res)
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
auto s1 = take(42);
auto s2 = transform(s1, f);
auto s3 = via(s2, gpu_executor{});
auto s4 = transform(s3, g);
s4.submit(receiver{&res});
Agenda

What are C++ executors?

Properties

Oneway executors

Twoway executors

Supporting affinity
P0796: Supporting Heterogeneous & Distributed Computing Through Affinity

High-level
P1436: Executor properties for affinity-based execution

Low-level
P1437: System topology discovery for heterogeneous & distributed computing (WIP)
● All systems are inherently heterogeneous
  ○ Desktop systems commonly have compute capable co-processors design for specific tasks such as GPUs or FPGAs
  ○ Server systems commonly have multiple CPU nodes or CPU + {GPU, FPGA, DSP, TPU, etc} nodes
  ○ Mobile and embedded SoC systems commonly have GPUs and/or often other specialised co-processors

● Many systems are distributed
  ○ HPC server and cloud systems have a distribution of a large number of interconnected nodes
  ○ These nodes can be connected physically or via network communication
• The structure of memory is no longer simple
  ○ Distributed memory regions across NUMA nodes
  ○ Hierarchical GPU memory regions
  ○ On-chip shared memory
  ○ Off-chip DMA transfers
  ○ Shared virtual memory through cache coherency
  ○ High Bandwidth Memory (HBM)
  ○ Persistent memory

• Memory allocation has to be adjusted to gain performance
  ○ Utilisation of shared memory regions (physical or virtual)
  ○ First touch memory allocation for lower latency access
  ○ Migration of memory allocations between discrete memory regions
● Define an interface for discovering and querying affinity
  ○ Solution must allow querying affinity related properties of an executor
  ○ Solution must provide process and memory affinity binding

● Integrate closely with the unified executors proposal (P0443)
  ○ Solution must align closely with the direction of the executors design

● Ensure scalability to heterogeneous and distributed systems
  ○ Solution needs to consider the limitations of heterogeneous and distributed systems to ensure scalability for future hardware
The property **bulk_execution_affinity** requires that an executor provide a guaranteed affinity binding pattern

- Pattern can be **none, balanced, scatter** or **compact**
- Requires that each execution agent be bound to a particular execution resource before the callable is called
- Binding must be consistent across all invocations of **bulk_execute** with the same size
<table>
<thead>
<tr>
<th></th>
<th>Socket 0</th>
<th></th>
<th>Socket 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td>0</td>
<td>Core 0</td>
<td>0</td>
</tr>
<tr>
<td>Core 1</td>
<td>1</td>
<td>Core 1</td>
<td>1</td>
</tr>
<tr>
<td>Core 0</td>
<td>2</td>
<td>Core 0</td>
<td>0</td>
</tr>
<tr>
<td>Core 0</td>
<td>3</td>
<td>Core 1</td>
<td>3</td>
</tr>
</tbody>
</table>
auto exec = execution::execution_context{execRes}.executor();

auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity.none);

affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
}, 8, sharedFactory);
auto exec = execution::execution_context{execRes}.executor();

auto affExec = execution::require(exec, execution::bulk,
   execution::bulk_execution_affinity.scatter);

affExec.bulk_execute([](std::size_t i, shared s) {
   func(i);
}, 8, sharedFactory);
auto exec = execution::execution_context{execRes}.executor();

auto affExec = execution::require(exec, execution::bulk,
    execution::bulk_execution_affinity::compact);

affExec.bulk_execute([](std::size_t i, shared s) {
    func(i);
}, 8, sharedFactory);
auto exec = execution::execution_context{execRes}.executor();

auto affExec = execution::require(exec, execution::bulk,
  execution::bulk_execution_affinity.balanced);

affExec.bulk_execute([](std::size_t i, shared s) {
  func(i);
}, 8, sharedFactory);
The query-only property **concurrency** returns the maximum potential concurrency available to it
- This provides a guide to the optimal bulk execution shape, but not a guarantee that
The query-only property `concurrency` returns the maximum potential concurrency available to it

- This provides a guide to the optimal bulk execution shape, but not a guarantee that

```cpp
executor exec;

size_t maxConcurrency = execution::query(exec, execution::concurrency);
```
• The query-only property `execution_locality_intersection` returns the maximum potential concurrency available to both of two executors
  ○ Tells you whether two executors will be contesting for the same resources
The query-only property `execution_locality_intersection` returns the maximum potential concurrency available to both of two executors:

- Tells you whether two executors will be contesting for the same resources

```cpp
executor_a execA;
executor_b execB;

size_t concurrencyOverlap = execution::query(execA,
                                           execution::execution_locality_intersection(execB));
```
• The query-only property `memory_locality_intersection` returns whether two execution resources share the same memory locality
  ○ Tells you whether memory allocated in each of the executors is in the same locale
The query-only property `memory_locality_intersection` returns whether two execution resources share the same memory locality:

- Tells you whether memory allocated in each of the executors is in the same locale.

```cpp
executor_a execA;
executor_b execB;

bool concurrencyOverlap = execution::query(execA,
    execution::memory_locality_intersection(execB));
```
Conclusions
● Executors did not make C++20
  ○ It was decided that some features need more time to bake before being ready for the standard

● So targeting C++23, what will executors look like?
  ○ There will be a properties mechanism, likely as seen in P0443
  ○ There will be oneway “fire and forget” executors, likely as seen in P0443
  ○ There will be twoway “sender/receiver” executors, likely to be along the lines of P1341
  ○ We hope there will be properties for affinity based allocation and execution, along the lines of P1436
• Some useful links:
  ○ Sender/receiver executors - http://wg21.link/p1341
  ○ Executor properties for affinity - https://wg21.link/p1436
Thanks for listening